On entity resolution in probabilistic data

Ayat, S.N.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).
4.1 Introduction

As discussed in Chapter 1, the concept of entity resolution (ER) is defined differently in different literature. In this chapter, we focus on the ER’s definition related to *identity resolution* (see Definition 1.2.1).

In chapters 1 and 3, we discussed the problem of entity resolution over probabilistic data (ERPD), which arises in many applications that deal with probabilistic data. In many of these applications, probabilistic data is distributed among a number of nodes. Let us give two examples of such applications, one from image retrieval, and the other from scientific data management domains.

4.1.1. Example. *Matching images in facial image databases.* For investigation purposes, consider that the police keeps a database of facial images of all persons who leave the country. In order to gather such data, suppose that the police has installed supervenience video cameras in all ports including airports, seaports, and land border ports. In each port, video cameras capture the video of the persons who leave the country. A face recognition system is then used to extract each individual’s facial image from captured videos, and store its feature vector as a tuple in a local database at the port. However, since the detected face might be moving in the video, and there is also an inherent uncertainty related to automated face recognition methods, each feature vector is associated with a probability value that represents its degree of certainty (e.g. as in [146]). Local databases are connected through a distributed system, which provides a query interface at each port for querying the set of all databases. At the same time, when a witness of a crime scene describes the facial features of a perpetrator, he/she is usually is not completely certain about some of the features. Therefore, the

---

1The material of this chapter has been partially published in [22] and [16].
police typically represents such a perpetrator’s face using an uncertain entity, say \( e \), consisting of a number of alternative feature vectors each of which is associated with a confidence value. An interesting question for the police is to identify if this perpetrator has left the country through any port, and thus querying the distributed database of all ports, in order to find the person who is most probably the same person as \( e \), where \( date > x \) (i.e. the date of the crime).

4.1.2. Example. Finding astronomical objects in astrophysics data. In astrophysics, as well as other scientific disciplines, the correlation and integration of the gathered observational data is the key for gaining new scientific insights. Astronomical observatories, distributed all over the world, produce data about sky surveys, some of which is uncertain [104]. In its simplified form, suppose each observatory maintains a single uncertain relation called \( Objects \), which contains data about the observed astronomical objects in its sky surveys. Each object in such a relation is then represented using a number of alternative tuples, each with a membership probability, showing its degree of certainty. However, the alternatives are mutually exclusive, meaning that at most one of them can be true. The uncertainty model, which is used in this example, has been already used in the database literature for representing astrophysics data [122]. Astrophysics researchers who want to gather information about a particular astronomical object, supposedly represented by an uncertain entity \( e \), are very interested in querying the astronomical objects observed in one sky’s region, in all distributed observatories, in order to find the object which is most probably the same object as the given object \( e \).

A straightforward approach for answering the above two queries is to ask all distributed nodes to send their data to a central node that can then deal with the problem of ERPD, by using one of the methods that are presented in Chapter 3. However, this approach is very expensive and does not scale well, neither with the size of databases, nor with the number of nodes. Therefore, using a distributed algorithm for dealing with the ERPD problem over distributed data is inevitable.

In this chapter, we propose the FD (Fully Distributed), which is a decentralized algorithm for dealing with the ERPD problem over distributed data, with the goal of minimizing the bandwidth usage and reducing the processing time. To the best of our knowledge, FD is the first proposal that deals with the ERPD problem over distributed data. It has the following salient features. First, it uses the novel concepts of Potential and essential-set to prune data at local nodes. This leads to a significant reduction of bandwidth usage compared to the baseline approaches. Second, its execution is completely distributed and does not depend on the existence of any certain node. We have validated FD through both implementation over a 75-node cluster and simulation. We have used both synthetic and real-world data in our experiments. The results show very good performance, in terms of bandwidth usage and response time.
4.2 Problem definition

In this section, we precisely define the problem addressed in the chapter.

For representing an uncertain database, we use the x-relation probabilistic data model [10] in which each uncertain entity is represented with an x-tuple (see the definition in Section 2.1.2). We denote an uncertain database by $D$, the set of its possible worlds by $PW(D)$, and the set of all tuples in $D$ by $\mathcal{D}$.

We assume that the uncertain database is fragmented over a number of nodes in a distributed system. We make no specific assumption about the topology of the distributed system architecture, which can be very general, e.g. an unstructured P2P system or a cluster. In the distributed system, each node knows some other nodes, i.e. its neighbors, to communicate with.

We define the problem of entity resolution for distributed probabilistic data as follows. Let $e$ be an uncertain entity issued at a query originator $p$. Let $TTL$ (Time To Live) determine the maximum hop distance which the user wants the entity resolution message to travel. Let $\mathcal{D}$ be the union of the uncertain databases that are in the schema of $e$, and maintained by nodes that can be accessed through $TTL$ hops from the query originator. Let $Sim$ be a context-free similarity function (refer to Section 3.2.2 for definition) for computing the similarity between tuples. Our goal is to find the most probable matching pair of $e$ and $\mathcal{D}$, i.e. $MPMP(e, \mathcal{D})$ (see Definition 3.2.2), while minimizing the communication cost.

4.3 Distributed Computation of Most-probable Matching-pair

One possible approach for computing MPMP is to move all relevant data of nodes to a central node, e.g. the query originator, where MPMP is computed using a centralized algorithm. However, the problem with this approach is that the query originator becomes a communication bottleneck, since it must receive a large amount of data from other nodes. In addition, it becomes a processing bottleneck, as it must process a large amount of data. In this section, we propose a fully distributed algorithm called FD, for computing MPMP. Our algorithm avoids the problems of the centralized approach by: 1) pruning all data that has
no chance to be MPMP, thus reducing the communication cost significantly; and
2) distributing the processing of MPMP over a large number of nodes.

4.3.1 Algorithm Overview

The FD algorithm starts at the query originator, the node at which a user issues
a query involving an uncertain entity $e$ to be resolved. The query originator
performs some initialization. First, it sets TTL to a value which is either specified
by the user or default, as found sufficient by the system in previous calculations.
Second, it gives $e$ a unique identifier, denoted by $eid$, which is made of a unique
node-ID and a query counter managed by the query originator. Nodes use $eid$ to
distinguish between new queries and those received before. After initialization, $e$
is included in a message that is then broadcasted by the query originator to its
reachable neighbors. Next, the entity resolution proceeds in the following phases
done at each node that receives the query:

- **Query forward.** Each node $p$ that receives the message including $e$ from
  a node $q$ performs the following steps. If it is the first time of receiving
  the query, then the node $p$ saves the id of $q$ as its parent, else-discards the
  message and makes a new message including $eid$ and sends it to $q$ to indicate
  that the query has been received from another node. Then $p$ decrements
  TTL by one, if TTL $\geq 0$, it makes a new message including $e$, $eid$, and new
  TTL; sends the message to all neighbors except $q$; and saves the number of
  sent messages to the neighbors.

- **Extract the essential-set.** The core idea of this phase is that for computing
  the most probable matching pair, the query originator does not need all
  entity-tuple pairs maintained at $p$, but only a subset of them that we call
  essential-set. In this phase, $p$ extracts the essential-set of its local data and
  saves it locally until receiving the essential-sets of its neighbors to which it
  has sent the query.

- **Merge-and-backward essential-sets.** In this phase, $p$ unifies its essential-
  set with those received from its neighbors into a set of entity-tuple pairs
  $essential_{pq}$, and sends $essential_{pq}$ to its parent, the node from which it
  received the query.

- **MPMP computation and data retrieval.** During the first three phases
  of the algorithm, the query originator receives a number of merged essential-
  sets from its neighbors. It unifies these sets with its local essential-set into
  the set $essential_{unified}$, and computes MPMP($e$, $\mathcal{D}$) and asks the node which
  contains the data to return the data content.

In the next subsections, we describe in more details the FD algorithm phases.
4.3.2 Extract the Essential-set

At each node \( p \), our FD algorithm prunes the data that have no chance to be the (global) most probable matching pair, i.e. MPMP(e, \( \mathcal{D} \)). For this, FD needs to extract the essential-set of each node which we define as follows. Let e be the given entity. Suppose \( D_p \) is the database maintained by \( p \) and \( n_p \) is the number of tuples in \( D_p \). Let \( S_p \) be the set of all entity-tuple pairs at \( p \), i.e. \( S_p = e \times D_p \). We define the essential-set of \( S_p \), denoted by \( \text{essential}(S_p) \) by using its complement: \( \text{essential}(S_p) \) is a subset of \( S_p \) whose members can never be MPMP(e, \( \mathcal{D} \)).

The alternatives of \( e \) are mutually exclusive, thus to find \( \text{essential}(S_p) \), it is sufficient to compute the essential-set for each alternative \( t \in e \), and then unify the essential-sets of all alternatives of \( e \). More precisely, we have:

\[
\text{essential}(S_p) = \bigcup_{t \in e} \text{essential}(S_{p,t}), \text{where } S_{p,t} = \{t\} \times D_p
\]

Now, we consider an alternative \( t \in e \), and explain the process of finding \( \text{essential}(S_{p,t}) \).

Let \( L_p = \{(t, t_{p,1}), \ldots, (t, t_{p,n_p})\} \) be the list of \( S_{p,t} \) pairs sorted in decreasing order of the similarities between \( t \) and \( D_p \) tuples. In other words, we have:

\[
\text{Sim}(t, t_{p,1}) > \cdots > \text{Sim}(t, t_{p,n_p}),
\]

where \( \text{Sim} \) is the given similarity function.

In the FD algorithm, we need to merge entity-tuple pair lists from other nodes with the pairs in list \( L_p \). Let \( \rho = (t, t_q) \) be an entity-tuple pair, from a node other than \( p \), that should be merged with \( L_p \). The pair \( \rho \) may be inserted in any index of \( L_p \), say index \( i \in [1, n_p + 1] \), based on the similarity between \( t \) and \( t_q \). The question in pruning is whether this pair has any chance to be MPMP(e, \( \mathcal{D} \)) or not. The answer to this question depends on the value of \( P_{\text{msp}}(\rho, \mathcal{D}) \), i.e. the probability that the pair \( \rho \) is the most similar pair\(^2\) (see Definition 3.2.2 in Section 3.2.3). However, \( P_{\text{msp}}(\rho, \mathcal{D}) \) depends not only on the pairs that are at node \( p \), but also on the pairs of other nodes. Thus, we cannot compute the exact value of \( P_{\text{msp}}(\rho, \mathcal{D}) \) locally, but we can compute an upper bound on this value. We denote such upper bound as the Potential of the index \( i \) of list \( L_p \). More precisely,

\[
\text{Potential}(i) = \max P_{\text{msp}}(\rho, \mathcal{D})
\]

where \( i \in [1..n_p + 1] \).

\(^2\)Notice that \( P_{\text{msp}}(\rho, \mathcal{D}) \) is the global \( P_{\text{msp}} \) value of pair \( \rho \), while \( P_{\text{msp}}(\rho, D_p) \) is its local \( P_{\text{msp}} \) value at node \( p \). Generally, \( P_{\text{msp}}(\rho, D_p) \geq P_{\text{msp}}(\rho, \mathcal{D}) \).
4.3.1. **Lemma.** Let \( i \) be an index in range \([1..n_p + 1]\). Let \( Y \) be the set of \( x \)-tuples formed by considering correlations between the tuples \( \{t_{p,1}, \ldots, t_{p,i-1}\} \), then

\[
\text{Potential}(i) = P(t) \times \prod_{x \in Y} (1 - P(x)).
\]

**Proof.** Let \( S_t = t \times D \) and \( L = \{(t, t_1), \ldots, (t, t_n)\} \) be the list of \( S_t \) pairs sorted based on their similarity in descending order. Let \( \rho = (t, t_q) \) resides in the index \( j \) of list \( L \), i.e. \( L[j] = \rho \). Using equation (3.1) (refer to Section 3.3, we have

\[
P_{\text{msp}}(\rho, D) = P(t_q) \times P(t) \times \prod_{x \in X} (1 - P(x))
\]

(4.2)

where \( X \) is the set of \( x \)-tuples formed by considering correlations between the tuples \( t_1 \) to \( t_j \) while the \( x \)-tuple containing \( t_q \) is omitted from it. It is clear that the value of \( P(t_q) \) which maximizes RHS(4.2) is equal to one. The set of tuples \( t_1 \) to \( t_{j-1} \) can be partitioned into two sets \( T_1 \) and \( T_2 \), where \( T_1 = \{t_{p,1}, \ldots, t_{p,i-1}\} \) is a subset of \( D_p \) and \( T_2 \) is a subset of \( D - D_p \). Let \( X_1 \) and \( X_2 \) be the set of \( x \)-tuples formed by considering correlations between the tuples in \( T_1 \) and \( T_2 \), respectively. Since all members of an \( x \)-tuple reside within the same node, \( x \)-tuple set \( X \) in RHS(4.2) can be partitioned into two disjoint sets \( X = X_1 \) and \( X_2 \), and, setting \( P(t_q) \) to one, equation (4.2) can be rewritten as

\[
P_{\text{msp}}(\rho, D) = P(t) \times \prod_{x \in X_1} (1 - P(x)) \times \prod_{x \in X_2} (1 - P(x))
\]

(4.3)

Notice that since we set \( P(t_q) \) to one, no \( x \)-tuple can contain it. Set \( X_1 \) is fixed, but we can make any assumption about set \( X_2 \) to maximize RHS(4.3). Each \( x \)-tuple \( x \) in set \( X_2 \) reduces RHS(4.3) by the factor of \( 1 - P(x) \), thus, RHS(4.3) is maximized when \( X_2 = \emptyset \). In such case, RHS(4.3) is equal to the asserted value in the lemma. \( \square \)

4.3.2. **Corollary.** Potential is a monotonically decreasing function.

Intuitively, Corollary 4.3.2 says that the higher is the index, the lower is its potential.

Let \( \text{local\_max} \) be the maximum local \( P_{\text{msp}} \) value of pairs in list \( L_p \), i.e. \( \text{local\_max} = \max P_{\text{msp}}(\rho, D_p), \rho \in L_p \). We use \( \text{local\_max} \) to define the stop index of list \( L_p \) as the smallest index in \([1..n_p + 1]\) where

\[
\text{Potential}(\text{stop}) < \text{local\_max}.
\]

(4.4)

The following lemma provides the basis for pruning the pairs in list \( L_p \).

4.3.3. **Lemma.** Let \( \text{stop} \) be the stop index of list \( L_p \). Then,

\[
\forall i \in [\text{stop}, n_p], L_p[i] \neq \arg \max_{\rho \in L_p} P_{\text{msp}}(\rho, D).
\]
4.3. Distributed Computation of MPMP

**Proof.** Let \( j \) be the index of a pair in list \( L_p \) with maximum local \( P_{msp} \) value, i.e. \( P_{msp}(L_p[j], D_p) = \text{local}\_\text{max} \). Let \( i \) be an index in list \( L_p \), where \( i \in [\text{stop}, n_p] \). Let \( S_i = \{t\} \times D \) and \( L = \{(t, t_1), \ldots, (t, t_n)\} \) be the list of \( S_i \) pairs sorted based on their similarity in descending order. Let \( j' \) and \( i' \) respectively be the index of pairs \( L_p[j] \) and \( L_p[i] \) in list \( L \), i.e. \( L[j'] = L_p[j] \) and \( L[i'] = L_p[i] \). To prove the lemma, we show that

\[
P_{msp}(L[i'], D) < P_{msp}(L[j'], D)
\]  \hspace{1cm} (4.5)

We have

\[
P_{msp}(L[i'], D) = P(t) \times P(t_{i'}) \times \prod_{x \in X_{i',1}} (1 - P(x)) \times \prod_{x \in X_{i',2}} (1 - P(x))
\]  \hspace{1cm} (4.6)

where \( X_{i'} \) is the set of \( x \)-tuples formed by considering correlations between the tuples \( t_1 \) to \( t_{i'} \), while the \( x \)-tuple containing \( t_{i'} \) is omitted from it. The set of tuples \( t_1 \) to \( t_{i'} \) can be partitioned into two sets \( T_{i',1} \) and \( T_{i',2} \), where \( T_{i',1} = \{t_{p,1}, \ldots, t_{p,i-1}\} \) is a subset of \( D_p \) and \( T_{i',2} \) is a subset of \( D - D_p \). Let \( X_{i',1} \) and \( X_{i',2} \) respectively be the set of \( x \)-tuples formed by considering correlations between the tuples in \( T_{i',1} \) and \( T_{i',2} \), while the \( x \)-tuple containing \( t_{i'} \) is omitted from \( X_{i',1} \). Since all members of an \( x \)-tuple reside within the same node, \( x \)-tuple set \( X_{i'} \) in RHS(4.6) can be partitioned into two disjoint sets \( X_{i',1} \) and \( X_{i',2} \), and equation (4.6) can be rewritten as

\[
P_{msp}(L[i'], D) = P(t) \times P(t_{i'}) \times \prod_{x \in X_{i',1}} (1 - P(x)) \times \prod_{x \in X_{i',2}} (1 - P(x))
\]  \hspace{1cm} (4.7)

Since \( L[i'], L_p[i], (t, t_{i'}), \) and \( (t, t_{p,i}) \) refer to the same pair, equation (4.7) can be written as

\[
P_{msp}(L[i'], D) = P_{msp}(L_p[i], D_p) \times \prod_{x \in X_{i',2}} (1 - P(x))
\]  \hspace{1cm} (4.8)

Using the same notation, we can write \( P_{msp}(L[j'], D) \) as

\[
P_{msp}(L[j'], D) = P_{msp}(L_p[j], D_p) \times \prod_{x \in X_{j',2}} (1 - P(x))
\]  \hspace{1cm} (4.9)

Based on the definition of stop index, it is clear that \( \text{stop} > j \), thus yielding \( i > j \). Thus, \( i' > j' \) and we have

\[
(\forall x \in X_{j',2}, \exists y \in X_{i',2} | x \leq y) \Rightarrow \prod_{y \in X_{j',2}} (1 - P(x)) \leq \prod_{x \in X_{i',2}} (1 - P(y))
\]  \hspace{1cm} (4.10)

Moreover, we know that

\[
P_{msp}(L_p[j], D_p) = \text{local}\_\text{max} > P_{msp}(L_p[i], D_p)
\]  \hspace{1cm} (4.11)
Using (4.8), (4.9), (4.10) and (4.11), we have

\[ P_{msp}(L[i'], \mathcal{D}) < P_{msp}(L[j'], \mathcal{D}) \]  \hspace{1cm} (4.12)

Since \( L[i'] = L_p[i] \) and \( L[j'] = L_p[j] \), (4.12) implies that

\[ L_p[i] \neq \text{arg max}_{\rho \in L_p} P_{msp}(\rho, \mathcal{D}). \]

\[ \square \]

Intuitively, Lemma 4.3.3 says that among all pairs in list \( L_p \), one of the pairs before the stop index has the maximum global \( P_{msp} \) value.

### 4.3.4. Corollary

Let \( \text{stop} \) be the stop index of list \( L_p \). Then,

\[ \forall i \in [\text{stop}, n_p], L_p[i] \neq \text{MPMP}(e, \mathcal{D}). \]

Intuitively, Corollary 4.3.4 says that the pair at the stop index and any pair after it have no chance to be the most probable matching pair. Thus, \( \text{essential}(S_{p,t}) \) is the set of \( L_p \) pairs whose index is smaller than the stop index.

#### Algorithm

Algorithm 6 describes the details of the steps which are performed for finding \( \text{essential}(S_p) \). Steps 3-19 are repeated for every alternative of \( e \), say \( t \), and at each iteration compute \( \text{essential}(S_{p,t}) \). Step 3 computes set \( S_{p,t} \) and step 4 sorts its pairs based on the similarity between the pair elements in descending order according to similarity function \( \text{Sim} \), and stores the result in list \( L \). Steps 5-7 compute list \( T \) as the second elements of list \( L \) and do some initialization. Steps 9-16 are repeated until finding the \( \text{stop} \) index of \( L \), and in each iteration, they process the pair at index \( i \) of list \( L \). Steps 10-12 compute \( P_{msp}(L[i], \mathcal{D}_p) \) as the intersection of two independent probabilistic events: \( t \) occurs; and among tuples \( T[1] \) to \( T[i] \), only \( T[i] \) occurs. To calculate the probability of the latter event, step 10 considers correlation among tuples to group tuples \( T[1] \) to \( T[i] \) into the set of x-tuples \( Y \) and step 11 removes the x-tuple containing \( T[i] \) from \( Y \) and stores the result in x-tuple set \( X \). Steps 13-15 update the current maximum \( P_{msp} \) of the pairs which we have processed so far. Steps 16 computes \( \text{Potential}(i + 1) \) using the x-tuple set \( Y \) which has already been computed in step 10. Step 17 checks if all pairs in the list \( L \) have been processed or \( i + 1 \) is the \( \text{stop} \) index of \( L \). If the condition holds, then the algorithm stops processing list \( L \), else it continues by processing the next pair in \( L \). Step 19 adds pairs \( L[1] \) to \( L[\text{stop} - 1] \) to the \( \text{essential-set} \). Finally, step 21 returns the \( \text{essential-set} \).
Algorithm 6 finding the essential-set

Input:
- Entity $e$
- Database $D_p$
- Similarity function $Sim$

Output: essential($S_p$), where $S_p = e \times D_p$

1: essential $\leftarrow \emptyset$
2: for all $t \in e$ do
3: \quad $S_{p,t} \leftarrow \{t\} \times D_p$
4: \quad $L \leftarrow \text{Sort}(S_{p,t}, Sim)$
5: \quad $T \leftarrow \{t' \mid (t, t') \in L\}$
6: \quad local$_{max} \leftarrow -1$
7: \quad $i \leftarrow 0$
8: repeat
9: \quad $i \leftarrow i + 1$
10: \quad $Y \leftarrow \text{set of}$ $x$-tuples involved in $\{T[i], \ldots, T[|L|]\}$
11: \quad // removing the $x$-tuple containing $T[i]$
12: \quad $X \leftarrow Y - \{x \mid x \in Y \land T[i] \in x\}$
13: \quad $P_{msp} \leftarrow P(t) \times P(T[i]) \times \prod_{x \in X} (1 - P(x))$
14: \quad if $P_{msp} > \text{local$_{max}$}$ then
15: \quad \quad \quad local$_{max} \leftarrow P_{msp}$
16: \quad end if
17: \quad Potential $\leftarrow P(t) \times \prod_{x \in Y} (1 - P(x))$
18: until (Potential $< \text{local$_{max}$}) \lor (i = |L|)$
19: \quad stop $\leftarrow i + 1$
20: \quad essential $\leftarrow$ essential $\cup \{L[1], \ldots, L[\text{stop} - 1]\}$
21: end for
22: return essential

Example

Let us illustrate the process of extracting essential-set using an example. Consider the uncertain entity $e$ and the uncertain database $D_p$ (maintained at node $p$) shown in Figures 4.1(a) and 4.1(b) respectively. In this example, $D_p$ contains single-alternative $x$-tuples, and entity $e$ has only one alternative. The set of existing entity-tuple pairs in node $p$, i.e. set $S_p$, can be computed as $S_p = e \times D_p$.

To prune $S_p$, we sort its pairs based on their similarity in descending order. The resulted list, denoted by $L$, is shown in Figure 4.1(c).

The Potential of the first location in list $L$, i.e. Potential(1), is equal to the probability of the event that $t$ occurs, which is equal to $P(t) = 0.8$. The $P_{msp}$ of the first entity-tuple pair in $L$, i.e. $(t, t_{p,3})$, is equal to the probability of the event that $t$ and $t_{p,3}$ occur, thus, $P_{msp}((t, t_{p,3}), D_p)$ is equal to $P(t) \times P(t_{p,3}) = 0.08$. The Potential of the second location in list $L$, i.e. Potential(2), is equal to the
Chapter 4. ER for Distributed Probabilistic Data

<table>
<thead>
<tr>
<th>t</th>
<th>P(t)</th>
<th>L[i]</th>
<th>P_{msp}(L[i], D_p)</th>
<th>Potential(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_p,1</td>
<td>0.4</td>
<td>(t, t_p,3)</td>
<td>0.08</td>
<td>0.8</td>
</tr>
<tr>
<td>t_p,2</td>
<td>0.7</td>
<td>(t, t_p,7)</td>
<td>0.504</td>
<td>0.72</td>
</tr>
<tr>
<td>t_p,3</td>
<td>0.1</td>
<td>(t, t_p,8)</td>
<td>0.1944</td>
<td>0.216</td>
</tr>
<tr>
<td>t_p,4</td>
<td>0.2</td>
<td>(t, t_p,3)</td>
<td>0.00864</td>
<td>0.0216</td>
</tr>
<tr>
<td>t_p,5</td>
<td>0.9</td>
<td>(t, t_p,2)</td>
<td>0.00907</td>
<td>0.01296</td>
</tr>
<tr>
<td>t_p,6</td>
<td>0.8</td>
<td>(t, t_p,7)</td>
<td>0.00280</td>
<td>0.00389</td>
</tr>
<tr>
<td>t_p,7</td>
<td>0.7</td>
<td>(t, t_p,8)</td>
<td>0.00025</td>
<td>0.000311</td>
</tr>
<tr>
<td>t_p,8</td>
<td>0.9</td>
<td>(t, t_p,1)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>(t, t_p,7)</td>
<td>0.000062</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.1: An example of uncertain entity $e$, database $D_p$, and the pruning process.

The probability of the event that $t$ occurs but $t_{p,3}$ does not occur, which is equal to $P(t) \times (1 - P(t_{p,3})) = 0.72$. This means that the maximum possible value for $P_{msp}$ of an entity-tuple pair which comes in $L[2]$ is 0.72. Since the Potential is greater than the current maximum value of $P_{msp}$, i.e. 0.08, we continue processing the list.

The $P_{msp}$ of the second pair in $L$, i.e. $(t, t_{p,7})$, is equal to the probability of the event that $t$ and $t_{p,7}$ occur but $t_{p,3}$ does not occur, which is equal to $P(t) \times P(t_{p,7}) \times (1 - P(t_{p,3})) = 0.504$. The Potential of the third location in list $L$, i.e. Potential(3), is equal to the probability of the event that $t$ occurs but neither $t_{p,3}$ nor $t_{p,7}$ occurs, which is equal to $P(t) \times (1 - P(t_{p,3})) \times (1 - P(t_{p,7})) = 0.216$. At this point, we stop processing the list since the Potential is less than the current maximum value of $P_{msp}$, i.e. 0.504. Therefore, the stop index of $L_p$ is 3 and essential($S_p$) is equal to $\{(t, t_{p,3}), (t, t_{p,7})\}$. To provide better intuition, the $P_{msp}$ and Potential values for other pairs are also shown in Figure 4.1(c).

4.3.3 Merge-and-Backward Essential-Sets

After extracting its essential-set, each node $p$ waits for receiving the essential-sets of its children (the nodes to which $p$ has sent the query). After receiving the essential-set of its children (or after a default wait time), $p$ merges its essential-set with those received from its children into a set of entity-tuple pairs essential$_{pq}$, and sends it to its parent.

In order to minimize network traffic, nodes do not bubble up the data items of entity-tuple pairs (which could be large), but only some needed information about them. The information that is put in the sent essential-set for each entity-tuple pair $(t_i, t_j)$, $t_i \in e$, $t_j \in D_q$, is a vector $(i, a, j, x, s, p)$ where $i$ is the index of $t_i$ in $e$, $a$ is the address of node $q$ which owns $t_j$, $j$ is the index of tuple $t_j$ in the database $D_q$ maintained by $q$, $x$ is the $x$-tuple to which $t_j$ belongs, $s$ is the similarity score between $t_i$ and $t_j$, and $p$ is the probability of tuple $t_j$. 

4.3. Distributed Computation of MPMP

4.3.4 MPMP Computation and Data Retrieval

When the query originator receives its children’s essential-sets, it merges them with its local essential-set into the set $\text{essential}_{\text{unified}}$. Theorem 4.3.5 shows that $\text{essential}_{\text{unified}}$ contains all entity-tuple pairs which are needed for computing $\text{MPMP}(e, D)$.

4.3.5. THEOREM. The entity-tuple pairs in set $\text{essential}_{\text{unified}}$ are sufficient for computing $\text{MPMP}(e, D)$.

Proof. Let $S$ be the set of all entity-tuple pairs at nodes which receive the query, i.e. $S = e \times D$. We show that we do not need any entity-tuple pair $\rho = (t, t'), \rho \in S - \text{essential}_{\text{unified}}$ for computing $\text{MPMP}(e, D)$.

Let $L$ be the list of pairs in set $\text{essential}_{\text{unified}}$ which have alternative $t \in e$ as their first element, and sorted based on their similarity in descending order. Let $\text{stop}_p$ be the stop index of a node, say node $p$, which comes before the stop indices of other nodes in list $L$. Using Lemma 4.3.3, pairs which come at or after $\text{stop}_p$ in $L$, cannot be the pair of $L$ with maximum $P_{\text{msp}}$. Thus, the pair of $L$ with maximum $P_{\text{msp}}$ lies in the range $[1..\text{stop}_p - 1]$. Now, we show that there is no entity-tuple pair $\rho = (t, t'), \rho \in S - \text{essential}_{\text{unified}}$, which may come before $\text{stop}_p$ in list $L$, and thus, is needed for computing the pair of $L$ with maximum $P_{\text{msp}}$. Pair $\rho$ is either maintained at node $p$ or at a node other than $p$, say $q$. In the former case, $\rho$ comes after $\text{stop}_p$ in list $L$ since $\text{stop}_p$ is the stop index of node $p$. Also in the latter case, $\rho$ comes after $\text{stop}_p$ in list $L$ since $\rho$ comes after the stop index of $q$ which itself comes after $\text{stop}_p$ in list $L$. Thus, using the pairs $L[1]$ to $L[\text{stop} - 1]$, we can compute the pair with maximum $P_{\text{msp}}$ and thereby $\text{MPMP}(e, D)$. □

Algorithm 7 shows the detailed steps which the query originator performs to compute $\text{MPMP}(e, D)$. Notice that:

- $\text{current\_max}$ does not represent the maximum value of $P_{\text{msp}}$ of the pairs in one list, i.e. related to alternative $t \in e$, but the current maximum $P_{\text{msp}}$ value of the pairs which we have visited so far. Thus, we reset it only once in the beginning of the algorithm.

- We use $\text{Potential}$ to stop early in visiting the pairs of list $L$. Notice that we may discard a list of pairs altogether because the maximum possible $P_{\text{msp}}$ of the pairs in that list (i.e. $\text{Potential}(1)$) is less than the current maximum $P_{\text{msp}}$ value that we got so far.

- Since the set $S_t$ consists of a number of sorted lists, the sort function in step 8 uses the sort-merge algorithm to merge these sorted lists.
Algorithm 7 computing MPMP(e, D)

Input: Set of entity-tuple pairs essential unified
Output: MPMP(e, D)

1: current_max ← -1
2: for all t ∈ e do
3: Potential ← P(t)
4: S_t ← {(t, t') | (t, t') ∈ essential unified}
5: length ← |S_t|
6: i ← 1
7: while (Potential > current_max) ∧ (i ≤ length) do
8: L ← Sort S_t pairs based on their similarity
9: T ← {t' | (t, t') ∈ L}
10: Y ← set of x-tuples involved in {T[1], ..., T[i]}
11: X ← Y - {x | x ∈ Y ∧ T[i] ∈ x}
12: Pmsp ← P(t) × P(T[i]) × \prod_{x∈X}(1 - P(x))
13: if Pmsp > current_max then
14: current_max ← Pmsp
15: MPMP ← L[i]
16: end if
17: Potential ← P(t) × \prod_{x∈Y}(1 - P(x))
18: i ← i + 1
19: end while
20: end for
21: return MPMP

Using Algorithm 7, the query originator computes MPMP(e, D) and asks the node which contains it to return the data content which is then returned to the user.

In the next section, we provide an example of all phases of the FD algorithm.

4.4 FD Example

In this section, we illustrate the FD algorithm with an example.

Consider a network consisting of three nodes o, q, and p as shown in Figure 4.2(a) and let these nodes respectively contain D_o, D_q, and D_p single-alternative databases which are shown in Figures 4.2(c), 4.2(d), and 4.2(e), respectively. Suppose that the user submits entity e, shown in Figure 4.2(b), to the node o, then FD performs the following phases for computing MPMP(e, D), where D = D_o ∪ D_q ∪ D_p:

1. In the “query forward” phase, node o forwards e to node q, and node q forwards e to node p.
2. In the “extract the essential-set” phase, each node extracts its essential-set as shown in Figures 4.2(h), 4.2(g), and 4.2(f). In the figures, we abbreviate Potential as Po. Also, the pairs that are transferred to the essential-set, are shown in bold, and since FD stops at the stop index, the values of \( P_{\text{msp}} \) and Potential for the pairs after the stop index are not shown. Notice that the stop index of list \( L_q \) if equal to \(|L_q|+1\), thus all pairs in this list are transferred into the essential-set.

3. In the “merge-and-backward essential-sets” phase, node \( p \) sends its essential set, i.e. \( \text{essential}(S_p) \), to node \( q \). Then, \( q \) merges the received set with its own essential set, i.e. \( \text{essential}(S_q) \), into set \( \text{essential}_{pq} \) (shown in Figure 4.2(i)), and sends it to node \( o \).

4. In the “MPMP computation and data retrieval” phase, node \( o \) merges its essential set, i.e. \( \text{essential}_O \), with the received set from \( q \), i.e. \( \text{essential}_{pq} \), into set \( \text{essential}_{\text{unified}} \) whose members are shown in Figure 4.2(j). Then, node \( o \) computes the MPMP as shown in Figure 4.2(j). The computed MPMP is equal to \( (t, t_{p,4}) \), thus node \( o \) asks node \( p \) for the data of the tuple \( t_{p,4} \).

### 4.5 Analysis of Communication Cost

In this section, we analyze the communication cost of FD, and as we will see, it is relatively low. We measure the communication cost in terms of number of messages and number of bytes which should be transferred over the network in order to execute a query by our algorithm. The messages transferred can be classified as: (1) forward messages, for forwarding the query to nodes; (2) backward messages, for returning the essential-sets from nodes to the query originator; (3) retrieve message, to request and retrieve the MPMP. Let us first formalize the distributed system model that we use in our analysis.

#### 4.5.1 Distributed System Model

Let \( P \) be the set of the nodes in the distributed system. Let \( Q \) be an entity resolution query at the query originator \( p_o \), i.e. the node at which the query is issued. Let \( P_Q \subseteq P \) be a set containing the query originator and all nodes that receive \( Q \). We model the nodes in \( P_Q \) and the links between them by a graph \( G(P_Q, E) \) where \( P_Q \) is the set of vertices in \( G \) and \( E \) is the set of the edges. There is an edge \( p \rightarrow q \) in \( E \) if and only if there is a link between the nodes \( p \) and \( q \) in the distributed system. Two nodes are called neighbor, if and only if there is an edge between them in \( G \). The number of neighbors of each node \( p \in P_Q \) is called the degree of \( p \) and is denoted by \( d(p) \).
A peer $p \in P_Q$ may receive $Q$ from some of its neighbors. The first node, say $q$, from which $p$ receives $Q$, is the parent of $p$ in $G$, so $p$ is a child of $q$. A node may have some neighbors that are neither its parent nor its children.
4.5.2 Forward Messages

Forward messages are the messages that we use to forward \( Q \) to the nodes. According to the basic design of our algorithm, each node in \( P_Q \) sends \( Q \) to all its neighbors except its parent. Let \( p_o \) denote the query originator. Let \( G(P_Q, E) \) be a graph representing the distributed network, such that \( P_Q \) is the set of nodes and \( E \) is the set of links between the nodes. With our FD algorithm, each node \( p \in \{ P_Q - \{ p_o \} \} \), sends \( Q \) to \( d(p) - 1 \) nodes, where \( d(p) \) is the degree of \( p \) in \( G \). The query originator sends \( Q \) to all of its neighbors, in other words to \( d(p_o) \) nodes. Then, the sum of all forward messages \( m_{fw} \) can be computed as

\[
m_{fw} = d(p_o) + \sum_{p \in \{ P_Q - \{ p_o \} \}} (d(p) - 1)
\]

We can write \( m_{fw} \) as follows:

\[
m_{fw} = \left( \sum_{p \in \{ P_Q \}} (d(p) - 1) \right) + 1 = \left( \sum_{p \in \{ P_Q \}} d(p) \right) - |P_Q| + 1 \tag{4.13}
\]

We use the average degree of the graph \( G \), denoted by \( d(G) \), to simplify (4.13). \( d(G) \) is defined as the average degree of nodes in \( G \) and can be computed as

\[
d(G) = \frac{\sum_{p \in P_Q} d(p)}{|P_Q|}
\]

Substituting \( d(G) \) in (4.13), we have

\[
m_{fw} = (d(G) - 1) \times |P_Q| + 1
\]

From the above discussion, we can derive the following Lemma.

4.5.1. Lemma. The number of forward messages in the FD algorithm is \( (d(G) - 1) \times |P_Q| + 1 \).

Proof. Implied by the above discussion. \( \square \)

In our underlying applications, e.g. astronomy application, the average degree of nodes is low, that is each node is usually connected to a small number of other nodes. Thus, the total number of forward messages is not very high compared to the number of nodes. For example, if the average degree of the system is 4, i.e. \( d(G) = 4 \), then we have \( m_{fw} = 3 \times |P_Q| + 1 \).

Let \( b_t \) be the average size of a tuple in \( Q \) in bytes, and \( |Q| \) be the number of alternative tuples of \( Q \). Then, the total size of data transferred by forward messages, denoted by \( b_{fw} \), can be computed as \( ((d(G) - 1) \times |P_Q| + 1) \times |Q| \times b_t \).
4.5.3 Backward Messages

In the Merge-and-Backward phase, each node in $P_Q$, except the query originator, sends its merged essential-set to its parent. Therefore, the number of backward messages, denoted by $m_{bw}$, is $m_{bw} = |P_Q| - 1$.

In the query forward phase of the algorithm, nodes in $P_Q$ are arranged in a tree, called query-tree, with the query originator as its root. For our modeling, we assume that the query-tree is a k-ary tree (i.e. $k = d(G)$) in which the root has $k$ children and all intermediate nodes has exactly $k - 1$ children. Moreover, we assume that all leaves are at the same level. These assumptions, however, are mostly for illustration purposes. In practice, nodes are organized in arbitrary tree topologies.

Let $h$ be the height of the tree, with the root at level $l = 0$. The total number of nodes, i.e. $|P_Q|$, can be computed as $|P_Q| = \sum_{l=2}^{h} (k - 1)^l + k + 1$.

Let $S(l)$ be the total size of data transferred in the Merge-and-Backward phase by each node which resides in the level $l$ of the query-tree. Let $b_{es}$ be the average size of the essential-set of a node. In the Merge-and-Backward phase, each node at level $h$ of the query-tree, i.e. leaf nodes, sends its essential-set to its parent. Thus, $S(h) = b_{es}$. Also, each intermediate node at level $l$, $l \neq 0$, of the query-tree receives exactly $k - 1$ essential-sets from its children which reside at level $l + 1$; merges them with its essential-set; and send the merged essential-set to its parent. Thus, for each intermediate node we have $S(l) = (k - 1) \times S(l + 1) + b_{es}$; thereby yielding the following recurrence relation for $S(l)$:

$$S(l) = \begin{cases} (k - 1) \times S(l + 1) + b_{es} & \text{for } 0 < l < h \\ b_{es} & \text{for } l = h \end{cases}$$

By solving this recurrence relation, we have

$$S(l) = \frac{1 - (k - 1)^{h-l+1}}{1 - (k - 1)} \quad (4.14)$$

Since there are exactly $k \times (k - 1)^{l-1}$ nodes at level $l$, $0 < l \leq h$, thus the total data transfer of the Merge-and-Backward phase, denoted by $b_{bw}$, can be computed as:

$$b_{bw} = \sum_{l=1}^{h} (k \times (k - 1)^{l-1} \times S(l))$$

By substituting $k = d(G)$ and $S(l)$ from (4.14) into the above equation, $b_{bw}$ can be written as

$$b_{bw} = \frac{b_{es} \times d(G) \times \left(1 + (h \times (d(G) - 2) - 1) \times (d(G) - 1)^h\right)}{(2 - d(G))^2}$$
Let $b_{pa}$ be the size of an entity-tuple pair in bytes, and $\eta$ be the average number of entity-tuple pairs of the essential-set which have the same alternative of $Q$ as their first element. Then, $b_{es}$, i.e. the average size of the essential-set in each node, can be computed as $b_{es} = |Q| \times \eta \times b_{pa}$.

In Section 4.6, we show that $\eta$ is very small and almost independent from the number of tuples which are maintained at a node. However, $\eta$ is dependent to the correlation between the probability of the tuples and their similarity to $Q$’s alternatives.

Let us show with an example that $b_{bw}$ is not significant. Consider that 10,000 nodes receive $Q$ (including the query originator), thus $|P_Q| = 10,000$. Assume that $d(G) = 4$. Thus, the height of the query-tree, i.e. $h$, is equal to 8. Our experiments show that $\eta$ is about 2.2 when similarity and probability are not correlated. Consider $Q$ has two alternative tuples. Since the actual data contents of the entity-tuple pair $(t_i, t_j)$ is not transferred during the Merge-and-Backward phase, we set $b_{pa}$ to 23, i.e. 1 bytes for $i$, 4 bytes for $j$, 6 bytes for the address of the node in which $t_j$ is maintained, 4 bytes for the x-tuple to which $t_j$ belongs, 4 bytes for the similarity score of $t_i$ to $t_j$, and 4 bytes for the probability of $t_j$. As a result, $b_{bw}$ is less than 10 megabytes for a distributed system that contains 10,000 nodes.

### 4.5.4 Retrieve Messages

By retrieve messages, we mean the message sent by the query originator to request the MPMP and the message sent by the node owning the MPMP to return it. Therefore, the number of retrieve messages, denoted by $m_{rt}$, is $m_{rt} = 2$. The total size of data transferred by these messages, denoted by $b_{rt}$, can be computed as $b_{rt} = m_{rt} \times b_t$, where $b_t$ is the average size of a tuple.

### 4.6 Performance Evaluation

We evaluated the performance of FD through implementation and simulation. The implementation over a 75-node cluster was useful to validate our algorithm in a realistic experimental environment. The simulation allowed us to study the performance of our algorithm under various conditions.

The rest of this section is organized as follows. In section 4.6.1, we describe our experimental and simulation setup, and the algorithms used for comparison. In section 4.6.2, we evaluate the response time of FD. Section 4.6.3 presents the evaluation of communication cost based on the bandwidth usage and the number of exchanged messages among nodes. In Section 4.6.4, we present the result of applying FD on real data.
4.6.1 Experimental and Simulation Setup

In our implementation and simulation, we compare FD with two baseline algorithms. The first algorithm is a centralized algorithm which we call FC (Fully Centralized). With FC, all nodes that receive the query send their data to the query originator where the most probable matching pair is computed using a centralized algorithm. The details of the centralized processing by FC can be found in Chapter 3. The second comparing algorithm is denoted by SCC (Score Confidence Centralized). In SCC, every node \( q \) receiving uncertain entity \( e \) (as the query) extracts a list containing the information of all of its pairs, and sends the extracted list directly to the query originator for centralized processing. More precisely, the information that is put in the sent list for each entity-tuple pair \((t_i, t_j) \in e \times D_q\), is a vector \((i, a, j, x, s, p)\) where \(i\) is the index of \(t_i\) in \(e\), \(a\) is the address of node \(q\) which owns \(t_j\), \(j\) is the index of tuple \(t_j\) in the database \(D_q\) maintained by \(q\), \(x\) is the x-tuple to which \(t_j\) belongs, \(s\) is the similarity score between \(t_i\) and \(t_j\), and \(p\) is the probability of tuple \(t_j\).

We implemented FD, FC, and SCC in Java, and tested them using a cluster of 75 nodes connected by a 1-Gbps network. Each node of cluster has a dual-quad-core 2.4 GHz processor and 24 GB memory. We make each node act as a node in the distributed system described in Section 4.5.1. We determined the node neighbors using the topologies generated by the BRITE universal topology generator [2]. Thus, each node only is allowed to communicate with the nodes that are its neighbors in the topology generated by BRITE.

To study the scalability of FD far beyond 75 nodes and to play with various performance parameters, we implemented a simulator using the PeerSim simulation kernel [5] and the Java programming language. We use the event driven engine of PeerSim to be able to simulate the delay in sending messages and also the bandwidth of nodes. We assign a random delay, denoted by latency, to communication ports to simulate the delay for sending a message between two nodes in a real distributed system. Also, we assign an upstream and a downstream bandwidth to each node. To simulate a node, we use a PeerSim’s node that performs all tasks that must be done by a node for executing FD, FC, and SCC algorithms. We implemented each of the three algorithms as a protocol in PeerSim. We used PeerSim’s WireKOut topology generator that randomly selects \(k\) neighbors for each node in the network. We used undirected links between nodes and set \(k\) to 10.

The experimental and simulation parameters are listed in Table 4.1. Notice that bandwidth and latency parameters are used only in our simulation. Unless otherwise specified, we use the values in this table for our tests. Each node has a table \(R(data, sim, p)\) in which attribute \(data\) is a random real number with normal distribution with a mean of 1 KB (Kilobytes) and a variance of 16 KB, \(sim\) is a random real number in the interval \([0..1]\) with normal distribution with a mean of 0.5 and a variance of 0.04, \(p\) is a random real number in the interval \((0..1]\)
4.6. Performance Evaluation

Table 4.1: Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>data</em>: tuple’s data items size</td>
<td>Gaussian random, Mean = 1 KB, Variance = 16 KB</td>
</tr>
<tr>
<td><em>sim</em>: similarity score</td>
<td>Gaussian random, Mean = 0.5, Variance = 0.04</td>
</tr>
<tr>
<td><em>p</em>: probability</td>
<td>Gaussian random, Mean = 0.4, Variance = 0.04</td>
</tr>
<tr>
<td><em>N</em>: number of tuples at each node</td>
<td>Uniform random integer in range [4500..5500]</td>
</tr>
<tr>
<td><em>d_x</em>: x-tuple’s average alternatives</td>
<td>3 for R table and 2 for the query</td>
</tr>
<tr>
<td><em>Cor</em>: correlation between <em>sim</em> and <em>p</em></td>
<td>0</td>
</tr>
<tr>
<td><em>Downstream bandwidth</em></td>
<td>8 × Upstream bandwidth</td>
</tr>
<tr>
<td><em>Latency</em></td>
<td>Gaussian random, Mean = 200 ms, Variance = 100 ms</td>
</tr>
<tr>
<td><em>Number of nodes</em></td>
<td>10,000</td>
</tr>
<tr>
<td><em>TTL</em>: Time To Live</td>
<td>100</td>
</tr>
</tbody>
</table>

with normal distribution with a mean of 0.4 and a variance of 0.04. Attribute *data* represents the data item that is returned back to the user as the result of the query and its value simulates the size of the data item. Attribute *sim* is used for computing the similarity between the tuple and the tuples in the query, and attribute *p* is the confidence value of the tuple. We introduce a number of parameters to control the characteristics of the R table. The number of tuples in R is denoted by *N*. The average number of alternatives that an x-tuple can have is denoted by *d_x*. The correlation between *sim* and *p* is denoted by *Cor*. To generate each tuple in R, we use a normal distribution for generating attribute *data*, and a bivariate normal distribution with a given correlation for generating *sim* and *p* attributes. We repeat this process to generate *N* different tuples. Then, we generate *d* as a uniform random integer in [1, 2 × *d_x* − 1], and repeatedly pick *d* tuples at random and group them into an x-tuple; if their confidence values add up to more than 1, we relinquish them and take another set of tuples until we form a valid x-tuple. We repeat this process until we group all tuples in valid x-tuples. We generate the query needed for experiments, in the same way that we generated a valid x-tuple. As Table 4.1 shows, we use the following default values for *N*, *d_x*, and *Cor* unless otherwise specified. *N* is a random number, uniformly distributed over all nodes, which is greater than 4500 and less than 5500. We use the default value *d_x* = 3 for the database and *d_x* = 2 for the query, and we use the default value *Cor* = 0.

For our implementation, we generate the R table and the query in the same way as our simulation, except for the data item size which is no longer simulated by a real number but with an array containing data.

Unless otherwise specified, we use the following values for the other simulation parameters. The *upstream bandwidth* of nodes is a random number with normal distribution with a mean of 56 Kbps (Kilobits per second) and a variance of 32 Kbps. The *downstream bandwidth* of each node is set to a value equal to 8 times of its *upstream bandwidth*. The *latency* for sending messages between any two nodes is also a random number with normal distribution with a mean of 200 milliseconds and a variance of 100 milliseconds.
Running the simulator on a machine with 16 GB of memory, allows us to perform tests up to 10,000 nodes, after which the simulation data no longer fit in RAM and makes our tests difficult. This is quite sufficient for our tests. Therefore, the number of nodes of the system is set to be 10,000, unless otherwise specified.

In all of our tests, we set $TTL$ to a high value, i.e. 100, to be sure that all nodes receive the query although the maximum hop-distance to other nodes from the query originator is much less than 100 with the topology that we use for our distributed system.

We repeat each simulation 10 times with the same query but with a different random number seed and average the outcomes.

Figure 4.3: Response time vs. number of tuples (on cluster)

Figure 4.4: Response time vs. number of tuples

Figure 4.5: Response time vs. number of nodes (on cluster)

Figure 4.6: Response time versus number of nodes
4.6. Performance Evaluation

4.6.2 Response Time

Scale up

In this section we study the response time of distributed entity resolution by varying the number of tuples, i.e. \( N \), and the number of nodes. The response time is the time elapsed from submitting the query to a node to sending the result of the query to the user. The response time includes local processing time and data transfer time. To study the effect of different correlations between similarity and confidence values, we ran experiments using three different correlations, i.e. negative, zero, and positive correlations.

We used our implementation over the cluster to study how the response time increases with increasing the number of tuples in each node. Using implementation over the cluster, Figure 4.3 shows the response times of FD, FC, and SCC with \( N \) increasing up to 8,000. Using simulation, Figure 4.4 shows the response times of the three algorithms with \( N \) increasing up to 5,000 and the other simulation parameters set as in Table 4.1.

While FD significantly outperforms the other two algorithms, its response time is affected only very little with increasing \( N \). As we expected, the negative correlation between similarity and confidence increases the response time since it increases the size of the essential-set at each node, and this increases the response time. Using independent random variables for similarity and confidence, i.e. zero correlation, decreases the response time and the positive correlation even decreases it more. Different correlations between similarity and confidence do not have any impact on FC and SCC algorithms, since they always send the whole database or the extracted similarity-confidence pairs of the whole database respectively.

We also used our implementation over the cluster to study the effect of the number of nodes on response time. Using implementation over the cluster, Figure 4.5 shows the response times of FD, FC, and SCC with the number of nodes increasing up to 75 and the other experimental parameters set as in Table 4.1. Using simulation, Figure 4.6 shows the response times of the three algorithms with the number of nodes increasing up to 10,000 and the other simulation parameters set as in Table 4.1. FD always significantly outperforms the other two algorithms and the performance difference increases significantly in the favor of FD as the number of nodes increases. These figures show excellent scale up of FD since response time logarithmically increases with increasing the number of nodes. We also observe that negative correlation between similarity and confidence increases the response time, but zero and positive correlations decrease the response time. Also, the performance difference between different correlations increases as the number of nodes increases.

The experimental results correspond with the simulation results. However, the response time of implementation over the cluster is better than that of simulation
because the cluster has a high-speed network.

To sum up, the reason of excellent scalability of FD versus both the database size and the number of nodes is its distributed execution. In FC and SCC algorithms, a central node, i.e. the query originator, is responsible for query execution, and this makes them inefficient.

**Effect of Latency and Bandwidth**

In this section, we study the effect of latency and bandwidth on response time. In the previous simulation tests the latency and upstream bandwidth were normally distributed random numbers with mean values of 200 ms and 56 Kbps respectively. In this test, we vary the mean values of the latency and bandwidth and study their effects on response time. For both experiments on bandwidth and latency, we set both $N$ and the number of nodes to 5,000 and other simulation parameters set as in Table 4.1.

Figure 4.7 shows how response time decreases with increasing bandwidth. Increasing the bandwidth has strong, similar effect on all three algorithms. FD outperforms the other two algorithms for all tested bandwidths.

![Figure 4.7: Effect of average bandwidth on response time](image1)

![Figure 4.8: Effect of average latency on response time](image2)

Figure 4.8 shows how response time evolves with increasing latency. Latency has little effect on the FC and SCC algorithms, because in these algorithms the nodes return their results directly to the query originator, and do not bubble up the results. Although FD outperforms the other algorithms for all the tested values, high latency, e.g. more than 500 ms, has strong impact on it and increases its response time much. However, below 500 ms, latency does not have much effect on FD’s response time.
4.6.3 Communication Cost

In this section, we study the communication cost of FD. We measure the communication cost in terms of the number of bytes, which should be transferred on the network for processing a query \( Q \). We also measure the number of exchanged messages during the execution of an algorithms. To study the effect of different similarity-confidence correlations, we ran experiments using three different correlations, i.e., negative, zero, and positive correlations.

Figure 4.9 shows how communication cost evolves with the number of tuples in each node increasing up to 5,000 and the other simulation parameters set as in Table 4.1. This figure shows that FD significantly outperforms the other two algorithms. Moreover, while increasing \( N \) has strong effect on FC and SCC algorithms, it has a very little effect on FD. Also as in scalability experiments, negative correlation between similarity and confidence values increases the communication cost, and zero or positive correlation decreases it.

Figure 4.10 shows the number of messages exchanged during the execution of the three algorithms with the number of tuples in each node increasing up to 5,000 and the other simulation parameters set as in Table 4.1. This figure shows that the database size has no effect on the number of exchanged messages. Although FD exchanges more messages than the other two algorithms, since the sizes of these messages are much smaller than the sizes of the messages produced by the other two algorithms, FD’s communication cost is significantly smaller than theirs.

Figure 4.10 also shows that different similarity-confidence correlations has no effect on the number of exchanged messages in FD.

We ran experiments to compare the average size of the essential-set with the number of entity-tuple pairs which exist at a node. To measure the average
essential-set size, we calculated the sum of the essential-set of all nodes and divided it by the number of nodes. In these experiments, we used uncertain entities with exactly 2 alternatives for the query. Figure 4.11 shows how the average size of the essential-set (in number of entity-tuple pairs) changes with the number of entity-tuple pairs in each node (i.e. \(2 \times N\)) increasing up to 10,000 and the other simulation parameters set as in Table 4.1. This Figure shows that the correlation between similarity and confidence has a strong effect on the size of the essential-set. The average size of the essential-set is almost constant for positive and zero correlations, i.e. 2 and 4.4 pairs respectively, but the essential-set size increases from 19.8 to 32.4 pairs for the negative correlation. These observations indicate that the size of the essential-set is very small and almost independent from the number of entity-tuple pairs which exist at the nodes. This means that our pruning algorithm performs quite effectively.

![Figure 4.11: Essential-set size vs. number of pairs in each node](image1.png)

![Figure 4.12: Effect of the number of nodes on the communication cost](image2.png)

We also ran experiments to study the effect of the number of nodes on communication cost. Figure 4.12 shows the communication costs of the three algorithms with the number of nodes increasing up to 10,000 and the other simulation parameters set as in Table 4.1. As this figure shows, FD significantly outperforms the other two algorithms and the performance difference increases significantly in the favor of FD as the number of nodes increases. Again as we expect, negative correlation between similarity and confidence increases the communication cost, but zero and positive correlations decrease the communication cost.

Figure 4.13 shows the number of messages exchanged during the execution of the three algorithms with the number of nodes increasing up to 10,000 and the other simulation parameters set as in Table 4.1. This figure shows that increasing the number of nodes increases the number of messages in the three algorithms. The number of exchanged messages in FD is higher than the other two algorithms but, as we discussed earlier because of the small size of these messages,
4.6. Performance Evaluation

FD significantly outperforms the other algorithms based on communication cost. Again as we expect, different similarity-confidence correlations has no effect on the number of exchanged messages in FD.

4.6.4 Case Study on Real Data

In this section, we report the result of applying the three algorithms on real data. As real-world database, we used a facial image database which we extracted from video. We downloaded 900 videos tagged with the keyword “wedding ceremony” from YouTube\(^3\), and used 2 fps sampling method and the pittpatt software [6] to extract 5010 distinct facial images each associated with a confidence value, from the videos. Then, we used SIFT method [89] and the bag of words model [87] with a codebook of 250 visual words to represent each facial image with a vector containing 250 real numbers in range \(0..1\) each associated with a confidence value. We used the cosine similarity metric for measuring the similarity between vectors.

![Figure 4.13: Effect of number of nodes on the number of exchanged messages](image)

![Figure 4.14: Response time vs. number of tuples (real data)](image)

We set the number of nodes to 10; \(k\) to 5; and the other network parameters as in Table 4.1. In each experiment, we randomly selected one of the vectors as the query, and equally distributed \(N\) randomly selected vectors among the nodes in the network.

Figures 4.14 and 4.15, respectively, show how response time and communication cost increase with \(N\) increasing up to 5,000. With increasing \(N\) up to 5,000, Figure 4.16 compares the average size of the essential-set with the total number of pairs in each node. As we expected, the result of applying the algorithms on real data confirms the result we observe on synthetic data.

\(^3\)http://www.youtube.com
4.7 Analysis against related Work

Beyond the work on entity resolution presented in Chapter 2, nearest neighbor and top-k queries over distributed uncertain data are also relevant to our research.

To the best of our knowledge, the existing nearest neighbor proposals, e.g. [44, 82, 125, 143], need the uncertain data to be stored in a centralized database, and thus cannot deal with the problem in distributed setting.

Recently, there have been some proposals dealing with the problem of top-k query processing for distributed uncertain data [140, 86]. In [140], the authors present a top-k query processing system for a wireless sensor network in which sensor nodes are grouped into clusters, where cluster heads are selected to perform localized data processing and to report aggregated results to the base station. Cluster heads use a user-specified probability threshold to find a rank boundary for pruning data gathered from sensors, before reporting to the base station. In [86], the authors present a proposal for ranking queries for distributed uncertain data. They use the concept of expected score and approximate it to reduce the communication cost and also processing time.

Our work nevertheless differs from these two above proposals because our problem definition is completely different. We look for an entity-tuple pair with the maximum probability of being the most similar pair, while [140] looks for tuples which have a probability higher than a user-specified threshold to be in the query result, and [86] is a proposal for approximating the expected score of the query results and then ranking them.
4.8 Conclusion

In this chapter, we proposed FD, a decentralized algorithm for dealing with the entity resolution problem over distributed probabilistic data, with the goal of minimizing the bandwidth usage. FD uses the novel concepts of potential and essential-set to prune data at local nodes. This leads to a significant reduction in bandwidth usage and response time compared to the baseline approaches. FD requires no global information, and does not depend on the existence of certain nodes.

We validated the performance of FD through implementation over a 75-node cluster and simulation using a simulator which we implemented using the PeerSim simulation kernel and the Java programming language. The experimental and simulation results show that response time of FD increases logarithmically with increasing the number of nodes. The experiments and simulations also show that FD’s response time is almost independent from the size of the database in nodes. The results also show the excellent performance of FD, in terms of communication cost, compared with the two baseline algorithms, i.e. FC, and SCC.